Simplified, stable parallel merging

Jesper Larsson Träff
Faculty of Informatics, Institute of Information Systems, Research Group Parallel Computing
Vienna University of Technology (TU Wien)
Favoritenstaße 16, 1040 Vienna
Austria

March 20, 2012

Abstract

This note makes an observation that significantly simplifies previous parallel, two-way merge algorithms based on binary search and sequential merge in parallel. First, it is shown that the additional merge step of distinguished elements as found in previous algorithms is not necessary, thus simplifying the implementation and reducing constant factors. Second, by fixating the requirements to the binary search, the merge algorithm becomes stable, provided that the sequential merge subroutine is stable. The stable, parallel merge algorithm can easily be used to implement a stable, parallel merge sort.

For ordered sequences with n and m elements, $m \leq n$, the simplified merge algorithm runs in $O(n/p + \log n)$ operations using p processing elements. It can be implemented on an EREW PRAM, but since it requires only a single synchronization step, it is also a candidate for implementation on other parallel, shared-memory computers.

Keywords: Parallel merging, Parallel algorithms, Implementation, Shared-memory computers, PRAM.

1 Introduction

All parallel, two-way merge algorithms (see, e.g., [3, 4, 5, 7, 9, 10, 11]) for the case where the number of elements in the sequences to be merged is larger than the number of processing elements seem to build on the scheme found in, e.g. [10]: Binary search is used to divide the two input sequences into disjoint sequences that can be merged pairwise independently. Binary searches are performed in parallel for a small selection of *distinguished elements* from the two input sequences. A separate, parallel merge of distinguished and located elements is needed to determine pairs of subsequences to merge. Often, such algorithms are not naturally stable, and take extra space to be made stable. This note shows that the parallel merge of distinguished and located elements is not needed, and at the same time makes the merge algorithm stable without any additional space or time overhead. This significantly simplifies implementation, whether on a PRAM [6, 8] or on real hardware with, e.g., OpenMP [1]. On an EREW PRAM the simplified algorithm runs in $O(n/p + \log n)$ time steps where n is the size of the longest input sequence.

	x_0				x_1				x_2				x_3			x_4			x_5
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
$A[i]$:	0	0	1	1	1	2	2	2	4	5	5	5	5	5	6	6	7	7	
						$\bar{y}_0$			$\bar{y}_1$	$\bar{y}_2$							$\bar{y}_3$		$\bar{y}_4$
																			$\bar{y}_5$

	$\bar{x}_0$															
	$\bar{x}_1$						$\bar{x}_2$	$\bar{x}_3$	$\bar{x}_4$							$\bar{x}_5$
j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$B[j]$:	1	1	3	3	3	3	4	5	6	6	6	6	7	7	7	
	y_0			y_1			y_2			y_3			y_4			y_5

Figure 1: Two non-decreasing sequences A and B with $n = 18$ and $m = 15$ elements, respectively, divided into $p = 5$ consecutive blocks. For the starting elements x_i and y_j of the blocks the corresponding low and high *cross ranks* are shown, denoted as $\bar{x}_i$ and $\bar{y}_j$, respectively. The cross ranks from the A array illustrate four of the five cases for the merge step: x_0 (a), x_1 and x_2 (e), x_3 (b), and x_4 (c). The cross ranks $\bar{y}_0$ and $\bar{y}_3$ from B illustrate case (d). The algorithm identifies the following $2p = 10$ merge subproblems of disjoint sequences that can be handled in parallel: $A[0, \ldots, 3]$ is copied into $C[0, \ldots, 3]$, $A[4]$ is copied into $C[4]$, $A[8]$ is copied into $C[14]$, $A[12, \ldots, 14]$ and $B[7]$ are merged into $C[19, \ldots, 22]$, $A[15]$ and $B[8]$ are merged into $C[23, 24]$ (all by Step 3); and $B[0, \ldots, 2]$ and $A[5, \ldots, 7]$ are merged into $C[5, \ldots, 10]$, $B[3, \ldots, 5]$ is copied into $C[11, \ldots, 13]$, $B[6]$ and $A[9, \ldots, 11]$ are merged into $C[15, \ldots, 18]$, $B[9, \ldots, 11]$ and $A[16, 17]$ are merged into $C[25, \ldots, 29]$, $B[12, \ldots, 14]$ is copied into $C[30, \ldots, 32]$ (all by Step 4).

2 The merge algorithm

Let A and B be two non-decreasing (possibly non-disjoint and allowed to contain repeated elements) sequences ordered by a relation $\leq$ with n and m elements, respectively. Assume without loss of generality that $m \leq n$. The input sequences are stored in arrays indexed from 0, and a merged output sequence is to be delivered in an array C with $n + m$ elements. For convenience, assume that $A[-1] = -\infty, A[n] = \infty$, and similarly for array B. An implementation does not have to store or reserve space for these sentinel elements, though.

For some element x and array X let its *low rank*, denoted $\mathsf{rank\_low}(x, X)$, be the unique index $i, 0 \leq i \leq n$ such that

$$X[i-1] < x \leq X[i]$$

and its *high rank*, denoted $\mathsf{rank\_high}(x, X)$, be the unique index j such that

$$X[j-1] \leq x < X[j]$$

The low (resp. high) rank of an element $a = A[i]$ (resp. $b = B[i]$) in the array B (resp. A) will be referred to as the *cross rank* of a (resp. b) in B (resp. A). The correctness of the merge algorithm is based on the observation that cross ranks "do not cross". By this, cross ranks from A can be used to partition B and vice versa. Consider the cross ranks of two elements $A[x_i]$ and $A[x_{i+1}]$ with $x_i < x_{i+1}$ in B. The observation below states that the cross rank of any element in B between the cross ranks of $A[x_i]$ and $A[x_{i+1}] - 1$ will be between x_i and x_{i+1}. Cross ranks for selected elements of the two sequences are shown in Figure 1.

Observation 1 *Let $a = A[i]$ an element of A and let $j = \mathsf{rank\_low}(a, B)$ be its cross rank in B. The cross rank for any element $j' < j$ that comes before j in B is not after i, i.e., $\mathsf{rank\_high}(B[j'], A) \leq i$, and the cross rank for any element $j'' > j$ that comes after j in B is strictly after i, i.e., $\mathsf{rank\_high}(B[j''], A) > i$. In particular, for $i' = \mathsf{rank\_high}(B[j], A)$ it is the case that $i' > i$. The same holds* mutatis mutandis *for elements in B.*

To see this, consider the cross rank $j = \mathsf{rank\_low}(a, B)$ of an element $a = A[i]$, and let $j' < j$. Since j is the low rank of a in B, $B[j'] \leq B[j-1] < a \leq B[j]$, so therefore $\mathsf{rank\_high}(B[j'], A) \leq i$. For $j < j''$, $a \leq B[j] \leq B[j'']$ and therefore $\mathsf{rank\_high}([B[j''], A) > i$.

Both low and high ranks can be computed by suitably modified binary search in $O(\log n)$ steps. The low rank of $a = A[i]$ from A is the number of elements from B that must come before a in a stable merge of A and B in which all (repeated) elements a from A are ordered before elements a from B, that is the position of a in the stably merged output sequence is $i + \mathsf{rank\_low}(A[i], B)$. Conversely, the high rank of $b = B[j]$ in A is the number of elements from A that must come before b in a stable merge, that is the position of b in the stably merged output is $j + \mathsf{rank\_high}(B[j], A)$.

Now, let p be the number of processing elements. The input sequences A and B are divided into roughly equal sized, consecutive, contiguous *blocks* differing in size by at most one. The first $r = n \bmod p$ blocks of A will get $\lceil n/p \rceil$ elements, and the remaining blocks $\lfloor n/p \rfloor$ elements; similarly for the B array. The start index x_i of block i in A for each $0 \leq i < p$ is determined by

$$x_i \quad = \quad \begin{cases} i\lceil n/p \rceil & \text{for } i < r \\ i\lfloor n/p \rfloor + n \bmod p & \text{otherwise} \end{cases}$$

In addition, define $x_p = n$. The start indices y_i for the p blocks of B are defined similarly. Let k be some index in A (resp. B). Index k belongs to block i if either $k < r\lceil n/p \rceil$ and $\lfloor k/\lceil n/p \rceil \rfloor = i$, or $k \geq r\lceil n/p \rceil$ and $\lfloor (k - r\lceil n/p \rceil)/\lfloor n/p \rfloor \rfloor + r = i$. Computing a block start index x_i or y_i as well as determining the block to which a given index k belongs are thus all constant time operations. With this, stable, parallel merge is accomplished by the following steps:

1. Compute $\bar{x}_i = \mathsf{rank\_low}(A[x_i], B)$ for $0 \leq i < p$ by binary search in parallel. Also, let $\bar{x}_p = m$.
2. Compute $\bar{y}_j = \mathsf{rank\_high}(B[y_j], A)$ for $0 \leq j < p$ by binary search in parallel. Also, let $\bar{y}_p = n$.
3. Merge disjoint sequences from A and B in parallel by assigning a processing element to each x_i for $0 \leq i < p$ as follows:
   (a) If $\bar{x}_i = \bar{x}_{i+1}$ output $A[x_i, \ldots, x_{i+1} - 1]$ to $C[x_i + \bar{x}_i, \ldots]$.
   (b) If $\bar{x}_i \neq \bar{x}_{i+1}$ are in the same block j of B, and $\bar{x}_i \neq y_j$, merge $A[x_i, \ldots, x_{i+1} - 1]$ stably with $B[\bar{x}_i, \ldots, \bar{x}_{i+1} - 1]$ and output to $C[x_i + \bar{x}_i, \ldots]$.
   (c) If $\bar{x}_i$ and $\bar{x}_{i+1}$ are in different B blocks, with $\bar{x}_i$ in block j, $\bar{x}_i \neq y_j$ and $\bar{x}_{i+1} \neq y_{j+1}$, merge $A[x_i, \ldots, \bar{y}_{j+1} - 1]$ stably with $B[\bar{x}_i, \ldots, y_{j+1} - 1]$. Output to $C[x_i + \bar{x}_i, \ldots]$.
   (d) If $\bar{x}_i$ and $\bar{x}_{i+1}$ are in different B blocks, with $\bar{x}_i$ in block j, $\bar{x}_i \neq y_j$ and $\bar{x}_{i+1} = y_{j+1}$, merge $A[x_i, \ldots, x_{j+1} - 1]$ stably with $B[\bar{x}_i, \ldots, y_{j+1} - 1]$. Output to $C[x_i + \bar{x}_i, \ldots]$.
   (e) If $\bar{x}_i$ is in block j of B, $\bar{x}_i = y_j$, and $\bar{x}_i \neq \bar{x}_{i+1}$, output $A[x_i, \ldots, \bar{y}_j - 1]$ to $C[x_i + \bar{x}_i, \ldots]$.

4. Merge disjoint sequences from B and A in the same fashion by assigning a processing element to each y_j for $0 \leq j < p$.

The results of the binary searches are shown in Figure 1 which illustrates in particular the five cases for the merge steps. By the observation on cross ranks, all blocks and sequences are disjoint, and obviously partition the arrays. In particular, for case (c), the condition implies that $\bar{x}_{i+1} \geq y_{j+1}$ and therefore it holds that $\bar{y}_j \leq x_{i+1}$, such that the segment $A[x_i, \bar{y}_{j+1} - 1]$ falls entirely within block i of A. The exception where $\bar{x}_{i+1} = y_{j+1}$ is covered by case (d); by the observation it namely holds that $\bar{y}_{j+1} > x_{i+1}$. Correctness is therefore established. Stability in the sense that all elements a of A will be placed before elements a of B is also maintained by the use of low and high ranks, provided a stable sequential merge is used. Since all blocks contain $O(n/p)$ elements and the sequences determined by the cross ranks as in the cases (a) to (e) fall entirely within blocks, the number of steps for the parallel merge operations with p processing elements is likewise $O(n/p)$. The binary searches are done in parallel and take $O(\log n)$ operations. Determining which merge case applies entails determining the blocks of $\bar{x}_i$ and $\bar{x}_{i+1}$ and takes $O(1)$ operations. In summary:

Theorem 1 *Two ordered sequences A and B of length n and m, respectively, with $m \leq n$ can be merged stably in parallel in $O(n/p + \log n)$ operations using p processing elements. Only constant extra space in addition to the input and output arrays is needed.*

Synchronization is only required after the two binary search steps, before which the cross ranks $\bar{x}_i$ and $\bar{y}_j$ are conveniently stored in $p+1$ element arrays. The algorithm can trivially be implemented on a CREW PRAM. For implementation on an EREW PRAM, it is first observed that each merge of a block from A requires comparing only two locations, namely $\bar{x}_i$ and $\bar{x}_{i+1}$, so accessing indices $\bar{x}_i$ and $\bar{x}_{i+1}$ by the p processing elements in different steps suffices to eliminate concurrent reads. Start addresses of the arrays A, B, and C can be copied to the p processing elements in $O(\log p)$ steps by parallel prefix operations. The parallel binary searches can be pipelined to eliminate concurrent reads, and still the p searches can be done in $O(\log n)$ parallel time steps. Also this is a standard technique. A more processor efficient algorithm for EREW PRAM parallel binary search can be found in [2].

Overall, the algorithm is a considerable simplification of [5, 10] and other similar merging algorithms in that no merging of the sequences of distinguished elements is needed.

3 Applications and remarks

The stable parallel merge algorithm can be used to implement a stable, parallel merge sort that runs in $O(n \log n/p + \log p \log n)$ parallel time steps for n element arrays. As in [10] this is done by first sorting sequentially in parallel p consecutive blocks of $O(n/p)$ elements, and then merging the sorted blocks in parallel in $\lceil \log p \rceil$ merge rounds. In round i, $i = 1, ..., \lceil \log p \rceil$ there are at most $\lceil p/2^i \rceil$ pairs to be merged and possibly one sequence that just has to be copied. This can be accomplished either by grouping the processing elements into groups of 2^i consecutively numbered elements, or by modifying the merge algorithm to work in parallel on the $\lceil p/2^i \rceil$ pairs. The latter can easily be accomplished. Thus, a stable merge sort can be implemented with no extra space apart from input and output arrays.

The simplified merge algorithm is likewise useful for distributed implementation, e.g. on a BSP as in [4]; here the eliminated merge of p pairs of distinguished elements can save at least one expensive round of communication. Details are outside the scope of this note.

References

[1] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[2] D. Z. Chen. Efficient parallel binary search on sorted arrays, with applications. *IEEE Transactions on Parallel and Distributed Systems*, 6(4):440–445, 1995.

[3] F. Gavril. Merging with parallel processors. *Communications of the ACM*, 18(10):588–591, 1975.

[4] A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27(6):809–822, 2001.

[5] T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.

[6] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[7] J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. *Informatique Théoretique et Applications*, 27(4):295–310, 1993.

[8] J. Keller, C. W. Keßler, and J. L. Träff. *Practical PRAM Programming*. John Wiley & Sons, 2001.

[9] C. P. Kruskal. Searching, merging and sorting in parallel computation. *IEEE Transactions on Computers*, C-32(10):942–946, 1983.

[10] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[11] P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: algorithm and implementation results. *Parallel Computing*, 15(1-3):165–177, 1990.